Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 70 (2008) 111–126
Science of Computer Programming
www.elsevier.com/locate/scico

Dynamic analysis of Java program concepts for visualization and profiling

Jeremy Singer*, Chris Kirkham

School of Computer Science, University of Manchester, UK

Received 1 December 2006; received in revised form 1 February 2007; accepted 11 July 2007
Available online 22 October 2007

Abstract

Concept assignment identifies units of source code that are functionally related, even if this is not apparent from a syntactic point of view. Until now, the results of concept assignment have only been used for static analysis, mostly of program source code. This paper investigates the possibility of using concept information within a framework for dynamic analysis of programs. The paper presents two case studies involving a small Java program used in a previous research exercise, and a large Java virtual machine (the popular Jikes RVM system). These studies investigate two applications of dynamic concept information: visualization and profiling. The paper demonstrates two different styles of concept visualization, which show the proportion of overall time spent in each concept and the sequence of concept execution, respectively. The profiling study concerns the interaction between runtime compilation and garbage collection in Jikes RVM. For some benchmark cases, we are able to obtain a significant reduction in garbage collection time. We discuss how this phenomenon might be harnessed to optimize the scheduling of garbage collection in Jikes RVM.

Keywords: Concept assignment; Dynamic analysis; Jikes RVM

1. Introduction

This paper fuses together ideas from program *comprehension* (concepts and visualization) with program *compilation* (dynamic analysis). The aim is to provide techniques to visualize Java program execution traces in a user-friendly manner, at a higher level of abstraction than current tools support. These techniques should enable more effective program comprehension, profiling and debugging. The overall objective is an improvement in software engineering practice.

1.1. Concepts

Program concepts are a means for high-level program comprehension. Biggerstaff et al. [1] define a concept as 'an expression of computational intent in human-oriented terms, involving a rich context of knowledge about the world.'

* Corresponding author.
E-mail addresses: jsinger@cs.man.ac.uk (J. Singer), chris@cs.man.ac.uk (C. Kirkham).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2007.07.006

They argue that a programmer must have some knowledge of program concepts (some informal intuition about the program's operation) in order to manipulate that program in any meaningful fashion. Concepts attempt to encapsulate original design intention, which may be obscured by the syntax of the programming language in which the system is implemented. Concept *selection* identifies how many orthogonal intentions the programmer has expressed in the program. Concept *assignment* infers the programmer's intentions from the program source code. As a simple example, concept assignment would relate the human-oriented concept `buyATrainTicket` with the low-level implementation-oriented artefacts:

```
{   queue();
    requestTicket(destination);
    pay(fare);
    takeTicket();
    sayThankYou();
}
```

Often, human-oriented concepts are expressed using UML diagrams or other high-level specification schemes, which are far removed from the typical programming language sphere of discourse. In contrast, implementation-oriented artefacts are expressed directly in terms of source code features, such as variables and method calls.

Concept assignment is a form of reverse engineering. In effect, it attempts to work backward from source code to recover the 'concepts' that the original programmers were thinking about as they wrote each part of the program. This conceptual pattern matching assists maintainers to search existing source code for program fragments that implement a concept from the application. This is useful for program comprehension, refactoring, and post-deployment extension.

Every source code entity is part of the implementation of some concept. The granularity of concepts may be as small as per token or per line; or as large as per block, per method or per class. Often, concepts are visualized by colouring each source code entity with the colour associated with that particular concept. Concept assignment can be expressed mathematically. Given a set U of source code units $u_0, u_1, \ldots, u_n$ and a set C of concepts $c_0, c_1, \ldots, c_m$ then concept assignment is the construction of a mapping from U to C. Often the mapping itself is known as the concept assignment.

Note that there is some overlap between concepts and aspects. Both attempt to represent high-level information coupled with low-level program descriptions. The principal difference is that concepts are universal. Every source code entity belongs to some concept. In contrast, only some of the source code implements aspects. *Aspects* encapsulate implementation-oriented cross-cutting concerns, whereas *concepts* encapsulate human-oriented concerns which may or may not be cross-cutting. Section 2.4 develops this relationship.

Throughout this paper, we make no assumptions about how concept selection or assignment takes place. In fact, all the concepts are selected and assigned manually in our two case studies. This paper concentrates on how the concept information is applied, which is entirely independent of how it is constructed. However we note that automatic concept selection and assignment is a non-trivial artificial intelligence problem. For instance, Biggerstaff et al. describe a semi-automated design recovery system called DESIRE [1]. This uses a precomputed domain model and a connectionist inference engine to perform the assignment. Gold and Bennett describe a hypothesis-based system [2]. This applies information from a human-generated knowledge base to source code using self-organizing maps to assign concepts.

1.2. Dynamic analysis with concepts

To date, concept information has only been used for static analysis of program source code or higher-level program descriptions [1,3,4]. This work focuses on *dynamic analysis* of Java programs using concept information. Such a dynamic analysis relies on embedded concept information within source code and dynamic execution traces of programs. This paper discusses various techniques for encoding, extracting and applying this concept information.

1.3. Contributions

This paper makes four major contributions:

(1) Section 2 discusses how to represent concepts practically in Java source code and dynamic execution traces.
(2) Sections 3.2 and 3.3 outline two different ways of visualizing dynamic concept information.

```
public @interface Concept1 { }
public @interface Concept2 { }
...
@Concept1 public class Test {
    @Concept2 public void f() { ... }
    ...
}
```

Fig. 1. Example Java source code that uses annotations to represent concepts.

(3) Sections 3 and 4 report on two case studies of systems investigated by dynamic analysis of concepts.
(4) Section 5 describes how concepts are used to profile garbage collection behaviour within a virtual machine.

2. Concepts in Java

This section considers several possible approaches for embedding concept information into Java programs. The information needs to be apparent at the source code level (for static analysis of concepts) and also in the execution trace of the bytecode program (for dynamic analysis of concepts).

There are obvious advantages and disadvantages with each approach. The main concerns are:

- Ease of marking up concepts, presumably in source code. We hope to be able to do this manually, at least for small test cases. Nonetheless it has to be simple enough for straightforward automation.
- Granularity of concept annotations. Ideally we would like to place concept boundaries at arbitrary syntactic positions in the source code.
- Ease of gathering dynamic information about concept execution at or after runtime. We hope to be able to use simple dump files of traces of concepts. These should be easy for post-process with perl scripts or similar ones.
- Ease of analysis of information. We would like to use visual tools to aid comprehension. We hope to be able to interface to the popular Linux profiling tool Kcachegrind [5], part of the Valgrind toolset [6].

The rest of this section considers different possibilities for embedded concept information and discusses how each approach copes with the above concerns.

2.1. Annotations

Custom annotations have only been supported in Java since version 1.5. This restricts their applicability to the most recent JVMs, excluding many research tools such as Jikes RVM[1] [7].

Annotations are declared as special `interface` types. They can appear in Java wherever a modifier can appear. Hence annotations can be associated with classes and members within classes. They cannot be used for more fine-grained (statement-level) markup.

Fig. 1 shows a program fragment that uses annotations to represent concepts in source code. It would be straightforward to construct and mark up concepts using this mechanism, whether by hand or with an automated source code processing tool.

Many systems use annotations to pass information from the static compiler to the runtime system. An early example is the AJIT system from Azevedo et al. [8]. Brown and Horspool present a more recent set of techniques [9].

One potential difficulty with an annotation-based concept system is that it would be necessary to modify the JVM, so that it would dump concept information out to a trace file whenever it encounters a concept annotation at runtime.

2.2. Syntax abuse

Since the annotations are only markers, and do not convey any information other than the particular concept name (which may be embedded in the annotation name) then it is not actually necessary to use the full power of annotations.

[1] The latest versions of Jikes RVM (post 2.4.5) have added support for custom annotations. We plan to look into how this allows us to extend our approach.

```
public class Concept1 extends Exception {
}

public class Test {
    public void f() throws Concept1 { ... }
    ...
}
```

Fig. 2. Example Java source code that uses syntax abuse to represent concepts.

```
public class Test {

    public static final int CONCEPT1 = ...;

    public void f(int concept) { ... }
    ...
}
```

Fig. 3. Example Java source code that uses metadata to represent concepts.

Instead, we can use *marker* interfaces and exceptions, which are supported by all versions of Java. The Jikes RVM system [7] employs this technique to convey information to the JIT compiler, such as inlining information and specific calling conventions.

This information can only be attached to classes (whose reference marker interfaces in their `implements` clauses) and methods (whose reference marker has exceptions in their `throws` clauses). No finer level of granularity is possible in this model. Again, these syntactic annotations are easy to insert into source code. Fig. 2 shows a program fragment that uses syntax abuse to represent concepts in source code. However a major disadvantage is the need to modify the JVM to dump concept information when it encounters a marker during program execution.

2.3. Custom metadata

Concept information can be embedded directly into class and method names. Alternatively each class can have a special concept field, which would allow us to take advantage of the class inheritance mechanism. Each method can have a special concept parameter. However this system is thoroughly intrusive. Consider inserting concept information after the Java source code has been written. The concept information will cause wide-ranging changes to the source code, even affecting the actual API. Fig. 3 shows a program fragment that uses metadata to represent concepts in source code. This is an unacceptably invasive transformation. Now consider using such custom metadata at runtime. Again, the metadata will only be useful on a specially instrumented JVM that can dump appropriate concept information as it encounters the metadata.

2.4. Aspects

Aspect-oriented programming (AOP) [10] provides new constructs to handle cross-cutting concerns in programs. Such concerns cannot be localized within single entities in conventional programming languages. In AOP, cross-cutting concerns are encapsulated using *aspects*. Aspects are encoded in separate source code units, distinct from the rest of the program source code. The code contained in an aspect is known as *aspect advice*. A point in the program source code where a cross-cutting concern occurs is known as a *join point*. At some stage during the compilation process, the appropriate aspect advice is *woven* into the main program source code at the relevant join points. The set of all join points for a particular aspect is known as a *point cut*. Event logging is the canonical aspect. Note that this is similar to generating a dynamic execution trace of concepts. The rest of this section uses AspectJ [11], which is the standard aspect-oriented extension of Java.

Each individual concept can be represented by a single aspect. Point cuts specify appropriate method entry and exit events for concept boundaries. Aspect advice outputs concept logging information to a dynamic execution trace stream. Fig. 4 shows a program fragment that uses aspects to represent concepts in source code.

```
aspect Concept1 {
    OutputStream conceptStream = System.out;

    pointcut boundary():
        call (void f())
        /* may have other methods here */
        ;

     before(): boundary() {
        conceptStream.println(‘‘concept1 entry’’);
    }

     after(): boundary() {
        conceptStream.println(‘‘concept1 exit’’);
    }
}
```

Fig. 4. Example AspectJ source code that uses aspects to represent concepts.

One problem with aspects is that the join points (program points at which concept boundaries may be located) are restricted. They are more general than simply method entry and exit points, but there are still some constraints. The AspectJ documentation [11] gives full details. Another disadvantage is that aspects are not integrated into the program until compilation (or possibly even execution [12]) time. Thus when a programmer inspects the original source code, it is not apparent where concept boundaries lie. It is necessary to consider both aspect advice and program code in parallel to explore concepts at the source code level.

2.5. Custom comments

A key disadvantage of the above approaches is that concepts can only be embedded at certain points in the program, for specific granularities (classes and methods). In contrast, comments can occur at arbitrary program points. It would be possible to insert concept information in special comments, that could be recognised by some kind of preprocessor and transformed into something more useful. For instance, the Javadoc system supports custom tags in comments. This approach enables the insertion of concept information at arbitrary program points. A Javadoc style preprocessor (properly called a *doclet system* in Java) can perform appropriate source-to-source transformation.

We eventually adopted this method for supporting concepts in our Java source code, due to its simplicity of concept creation, markup and compilation. Fig. 5 shows a program fragment that uses custom comments to represent concepts in source code.

The custom comments can be transformed to suitable statements that will be executed at runtime as the flow of execution crosses the marked concept boundaries. Such a statement would need to record the name of the concept, the boundary type (entry or exit) and some form of timestamp.

In our first system (see Section 3) the custom comments are replaced by simple `println` statements and timestamps are computed using the `System.nanoTime()` Java 1.5 API routine, thus there is no need for a specially instrumented JVM.

In our second system (see Section 4) the custom comments are replaced by Jikes RVM-specific logging statements, which are more efficient than `println` statements, but entirely non-portable. Timestamps are computed using the IA32 `TSC` register, via a new ‘magic’ method. Again this should be more efficient than using the `System.nanoTime()` routine.

In order to change the runtime logging behaviour at concept boundaries, all that is required is to change the few lines in the concept doclet that specify the code to be executed at the boundaries. One could imagine that a more complicated code is possible, such as data transfer via a network socket in a distributed system. However note the following efficiency concern: One aim of this logging is that it should be unobtrusive. The execution overhead of concept logging should be no more than noise, otherwise any profiling will be inaccurate. In the studies described in

```
// @concept_begin Concept1
public class Test {
    public void f() {
        ....

        while (...)
            // @concept_end Concept1
            // @concept_begin Concept2
    }
    ...
}
// @concept_end Concept2
```

Fig. 5. Example Java source code that uses comments to represent concepts.

this paper, the mean execution time overhead for running concept-annotated code is 35% for the small Java program (Section 3) but only 2% for the large Java program (Section 4). This disparity is due to the relative differences in concept granularity in the two studies. All the experiments in this paper are based on *exhaustive tracing* of concept information. Note that a *statistical sampling* approach would require less overhead than exhaustive tracing. A concept sampler could be incorporated with the timer-based method profiler used in most adaptive JVM systems to identify frequently executed regions of code.

There are certainly other approaches for supporting concepts, but the five presented above seemed the most intuitive and the final one seemed the most effective.

3. Dynamic analysis for small Java program

The first case study involves a small Java program called `BasicPredictors` which is around 500 lines in total. This program analyses streams of ASCII characters encoding method return values. It computes how well these values could be predicted using standard hardware mechanisms such as last value prediction [13] and finite context method [14]. The program also computes information theoretic quantities such as the entropy of the value stream. We used this program to generate the results for an earlier study on method return value predictability for Java programs [15].

3.1. Concept assignment

The `BasicPredictors` code is an interesting subject for concept assignment since it calculates values for different purposes in the same control flow structures (for instance, it is possible to re-use information for prediction mechanisms to compute entropy).

We have identified four concepts in the source code.

system: the default concept. Prints output to stdout, reads in input file, reads arguments, allocates memory.
predictor_compute: performs accuracy calculation for several *computational* value prediction mechanisms.
predictor_context: performs accuracy calculation for *context-based* value prediction mechanism (table lookup).
entropy: performs calculation to determine information theoretic entropy of entire stream of values.

The concepts are marked up manually using custom Javadoc tags, as described in Section 2.5. This code is transformed using the custom doclet, so the comments have been replaced by `println` statements that dump out concept information at execution time. After we have executed the instrumented program and obtained the dynamic execution trace which includes concept information, we are now in a position to perform some dynamic analysis.

3.2. Dynamic analysis for concept proportions

The first analysis simply processes the dynamic concept trace and calculates the overall amount of time spent in each concept. (At this stage we do not permit nesting of concepts, so the code can only belong to a single concept

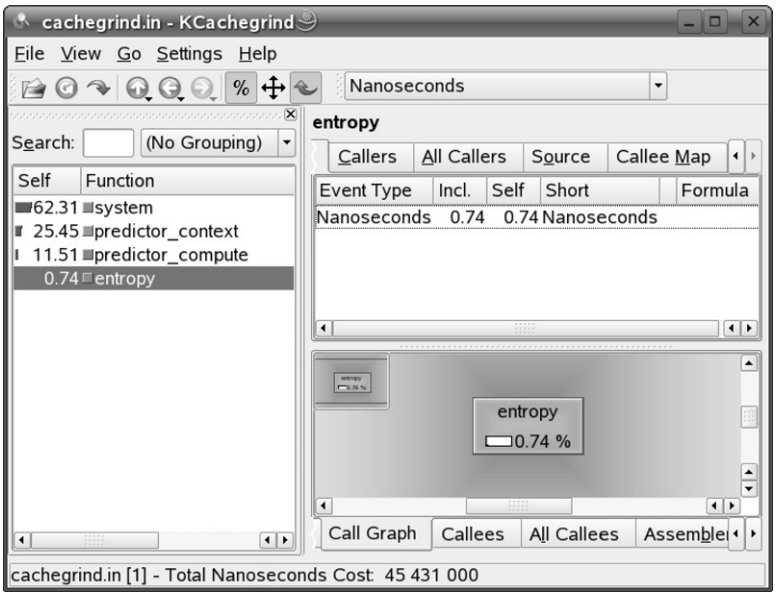

Fig. 6. Screenshot of Kcachegrind tool visualizing percentage of total program runtime spent in each concept.

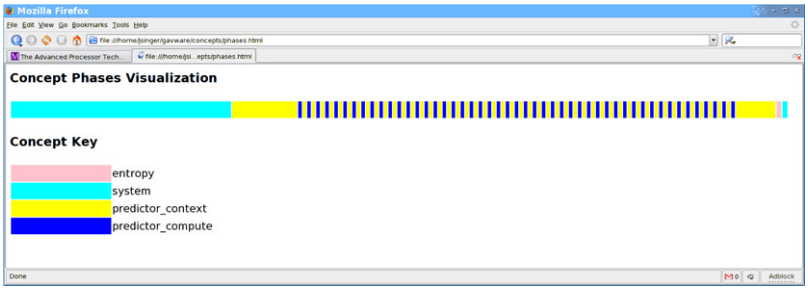

Fig. 7. Simple webpage visualizing phased behaviour of concept execution trace.

at any point in execution time.) This analysis is similar to standard function profiling, except that it is now based on specification-level features of programs, rather than low-level syntactic features such as function calls.

The tool outputs its data in a format suitable for use with the Kcachegrind profiling and visualization toolkit [5]. Fig. 6 shows a screenshot of the Kcachegrind system, with data from the `BasicPredictors` program. It is clear to see that most of the time (62%) is spent in the `system` concept. It is also interesting to note that `predictor_context` (25%) is more expensive than `predictor_compute` (12%). This is a well-known fact in the value prediction literature [14].

3.3. Dynamic analysis for concept phases

While this analysis is useful for determining the overall time spent in each concept, it gives no indication of the temporal relationship between concepts. It is commonly acknowledged that programs go through different phases of execution which may be visible at the micro-architectural [16] and method [17,18] levels of detail. It should be possible to visualize phases at the higher level of concepts also.

So the visualization in Fig. 7 attempts to plot concepts against execution time. The different concepts are highlighted in different colours, with time running horizontally from left to right. Again, this information is extracted from the dynamic concept trace using a simple perl script, this time visualized as HTML within any standard web browser.

There are many algorithms to perform phase detection but even just by observation, it is possible to see three phases in this program. The startup phase has long periods of `system` (opening and reading files) and `predictor_context`

Fig. 8. Key for concept visualizations.

(setting up initial table) concept execution. This is followed by a periodic phase of prediction concepts, alternately `predictor_context` and `predictor_compute`. Finally there is a result report and shut-down phase.

3.4. Applying this information

How can these visualizations be used? They are ideal for program comprehension. They may also be useful tools for debugging (since concept anomalies often indicate bugs [19]) and profiling (since they show where most of the execution time is spent).

This simple one-dimensional visualization of dynamic concept execution sequences can be extended easily. It would be necessary to move to something resembling a Gantt chart if we allow nested concepts (so a source code entity can belong to more than one concept at once) or if we have multiple threads of execution (so more than one concept is being executed at once).

4. Dynamic analysis for large Java program

The second case study uses Jikes RVM [7] which is a reasonably large Java system, around 300,000 lines of code. It is a production-quality adaptive JVM written in Java. It has become a significant vehicle for virtual machine (VM) research, particularly into adaptive compilation mechanisms and garbage collection. All the tests reported in this section use Jikes RVM version 2.4.4, development configuration, Linux/IA-32 build and single pthread VM runtime.

Like all high-performance VMs, Jikes RVM comprises a number of adaptive runtime subsystems, which are invoked on demand as user code executes. These include just-in-time compilation, garbage collection and thread scheduling. A common complaint from new users of Jikes RVM is that it is hard to understand how the different subsystems operate and interact. The *programmer* is not aware of how and when they will occur, unless he explicitly requests their services by sending messages like `System.gc()`, but this is rare. Similarly, the *user* is not aware of when these subsystems are operating, as the code executes. They are effectively invisible, from both a static and a dynamic point of view. So this case study selects some high-level concepts from the adaptive infrastructure, thus enabling visualization of runtime behaviour.

4.1. Visualized subsystems

After some navigation of the Jikes RVM source code, we inserted concept tags around a few key points that encapsulate the adaptive mechanisms of (i) garbage collection and (ii) method compilation. These are the dominant VM subsystems in terms of execution time. Other VM subsystems, such as the thread scheduler, have negligible execution times so we do not profile them. Note that all codes not in an explicit concept (both Jikes RVM code and user application code) are in the default `unmarked` concept. Fig. 8 gives the different concepts and their colours. Figs. 9–11 show different runs of the `_201_compress` benchmark from the SPECjvm98 suite, and how the executed concepts vary over time.

The *garbage collection* VM subsystem (GC) manages memory. It is invoked when the heap is becoming full, and it detects unreachable (*dead*) objects and deletes them, thus freeing up heap space for new objects. All our experiments use the default Jikes RVM generational mark-sweep garbage collection algorithm. This algorithm's distinct behaviour is clear to see from the concept visualizations. There are two generations: nursery and mature. The nursery generation is cheap to collect, so generally a small amount of GC time is spent here. On the other hand, the mature generation is more expensive to collect but collections are less frequent, so occasionally a longer GC time occurs. This is most apparent in Fig. 9. Most GCs are short but there is one much longer GC around 45% of the way through the execution.

Fig. 9. Pervasive nature of VM subsystem execution.

Fig. 10. Varying nature of VM subsystem execution, illustrated by two runs of the same benchmark program.

Fig. 11. Periodic nature of VM subsystem execution.

The *compilation* VM subsystem is divided between two concepts. The *baseline* compiler is a simple bytecode macro-expansion scheme. It runs quickly but generates inefficient code. On the other hand the *optimizing* compiler (optcomp) is a sophisticated program analysis system. It runs slowly but generates highly optimized code. Generally all methods are initially compiled with the baseline compiler, but then frequently executed (*hot*) methods are recompiled with the optimizing compiler. The time difference is clear in the visualizations. For instance, the bottom trace in Fig. 10 has many short baseline compilations and a few much longer optimizing compilations.

4.2. Subsystem properties

This section presents five VM subsystem properties that our concept visualizations clearly demonstrate. This enables us to gain a better understanding of VM behaviour in general.

4.2.1. Pervasive

VM subsystems are active throughout the entire program lifetime. Fig. 9 illustrates this point. It is noticeable that the density of VM code in relation to user code changes over time. It appears that VM code is more frequent near the beginning of execution. This is because the compiler compiles every method immediately before it is executed for the first time. Later compilation activity generally involves selective optimizing recompilation of hot methods.

4.2.2. Significant runtime

Visualizations like Fig. 9 show that VM code occupies a significant proportion of total execution time. The total execution time is shared between application code and VM code. For the long-running benchmark program used in this study, around 90% of time is spent in user code and 10% in VM code. The VM time should be more significant for shorter programs. It is also interesting to discover how the VM execution time is divided between the various subsystems. For the benchmark used in this study, the VM spends at least twice as long in compilation as in garbage collection.

4.2.3. Dependence on VM configuration

Modern adaptive runtimes are highly configurable. It is possible to specify policies and parameters for all subsystems. These have a major impact on system performance. Previously it was not possible to see exactly how varying configurations changed overall behaviour. Now our concept visualizations make this task straightforward. Fig. 10 shows two runs of the same program, but with different VM configurations. The top trace uses the default VM compilation policy, which at first compiles all methods using the baseline compiler then recompiles hot methods with the more expensive optimizing compiler. The bottom trace uses a modified compilation policy, which initially uses the optimizing compiler rather than the baseline compiler, as much as possible. Note that Jikes RVM requires that some methods must be compiled at baseline level.

4.2.4. Interactivity

VM subsystems are not entirely independent. They affect one another in subtle ways. For instance in Fig. 10, it is clear to see that greater use of the optimizing compiler causes increased garbage collection activity. This is because the optimizing compiler generates many intermediate program representations and temporary data structures as it compiles methods, thus filling up heap space. Note that there is a single heap shared between VM code and user code. Section 5 explores this interaction in more detail.

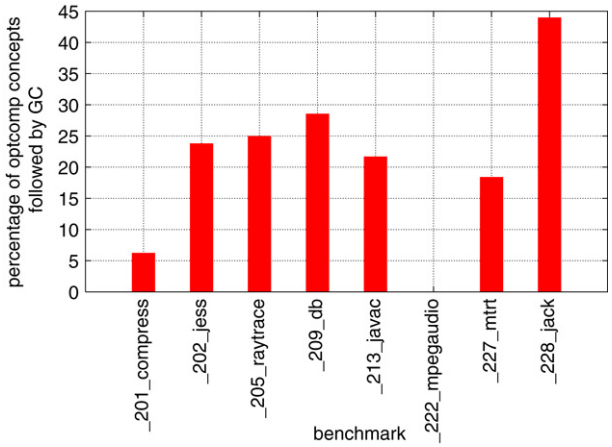

Fig. 12. Percentage of optcomp concepts that are followed by GC, for each benchmark program.

4.2.5. Periodicity

Programs go through different phases and exhibit periodic patterns, as Section 3.3 mentions. Fig. 11 demonstrates that VM code may be periodic too. In this execution trace, the program is memory intensive and the heap has been restricted to a small size. Thus the garbage collector has to run frequently, and if the memory load is constant over time, then the garbage collector will run periodically.

5. Profiling garbage collection

Section 4.2 noted that optimizing compilation (optcomp) frequently triggers GC. We assume this is because optcomp creates many intermediate data structures such as static single assignment form when it analyses a method. This takes up space in the heap. (VM and user code share the same heap in Jikes RVM system.) However most of these intermediate compilation objects die as soon as the compilation completes. A recent study [20] shows that it is most efficient to do GC when the proportion of dead-to-live data on the heap is maximal, even if the heap is not entirely full. Processing live data wastes GC time. Live data must be scanned and perhaps copied. On the other hand, dead data may be immediately discarded. This insight leads to our idea that it may be good to perform GC *immediately after optcomp*. Thus we consider modifying the VM to force GC automatically after every optimizing compilation.

We query a set of dynamic execution traces to determine how often GC follows optcomp, in a standard Jikes RVM system setup. We use the default GC strategy (generational mark-sweep) with standard heap sizes (50 MB start, 100 MB maximum). We gather concept data from the entire SPECjvm98 benchmark suite. For each benchmark, we measure the percentage of optcomp concepts that are followed by GC (i.e. GC is the next VM concept after optcomp, possibly with some intervening unmarked concept code). Fig. 12 shows the results. For some programs, around 25% of optcomp concepts are followed by GC. However the proportion is much lower for others. This suggests that any optimization should be program specific. Presumably since some methods have larger and more complex methods, optcomp has to do more work and uses more memory.

Now we modify the optimizing compiler so that it forces a GC immediately after it has completed an optcomp concept (eager GC-after-optcomp). We hope to target the heap when a large proportion of objects have just become dead. We use the Jikes RVM memory management toolkit harness code to measure the amount of time spent in GC throughout the entire execution of each benchmark. In order to stress the GC subsystem, we decide to use relatively small heap sizes for each benchmark. We determine the minimum fixed heap size (specified using the `-Xms -Xmx` options for Jikes RVM) in which each benchmark will run without throwing any out-of-memory exceptions. Note that within the fixed heap, the nursery generation may expand to fill the available free space. Fig. 13 shows these minimum heap sizes for each benchmark. We conducted three sets of experiments, using 1, 1.5 and 2 times the minimum heap size for each benchmark. All timing figures are taken as the median score of up to five runs.

In our preliminary experiments, we modified the Jikes RVM GC policy to force a collection immediately after each optcomp. However, we noticed that this actually causes a performance degradation. We changed the GC policy so that the VM checks to see if the heap usage has exceeded a certain threshold, immediately after each optcomp. If

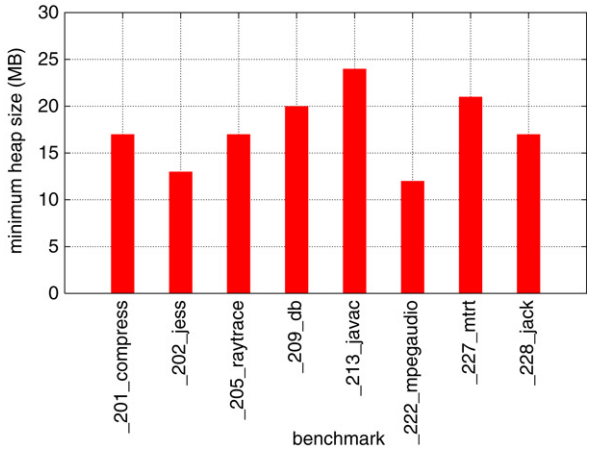

Fig. 13. Minimum possible Jikes RVM heap sizes for each benchmark program.

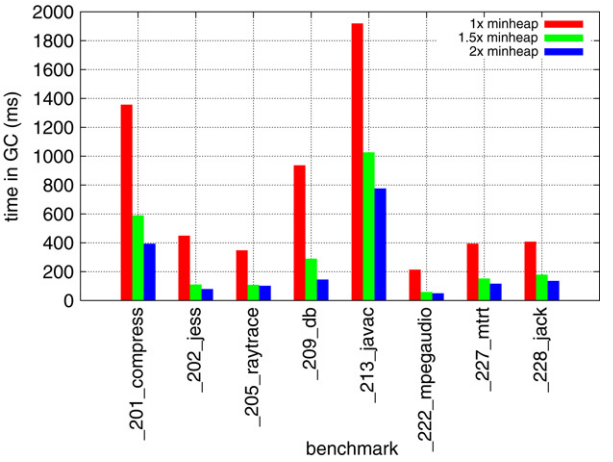

Fig. 14. GC times for different benchmarks before VM modification.

the threshold is exceeded, we force a collection. All the experiments below use this threshold-based eager GC-after-optcomp policy on modified VMs. We arbitrarily chose to set the threshold to 0.9. A more detailed profiling study would assess various threshold values to determine an optimal heuristic.

Fig. 14 shows the GC times for each benchmark. These initial experiments are run on the unmodified VM, so garbage collection only occurs when the standard VM heap usage monitoring code detects that the heap is nearly full. Then Fig. 15 shows the relative difference in GC times between the unmodified and modified VMs. A negative score indicates a speedup in the modified VM, whereas a positive score indicates a slow-down. There is a clear variation in performance, with the most obvious improvements occurring for the minimum heap size, in general.

Finally we investigate how this eager GC-after-optcomp strategy affects the overall runtime of the programs. Reduction in GC time has a direct impact on overall execution time, since GC time is included in the overall time. However, there is also an indirect impact caused by improved GC. The execution time of the benchmark code itself may be reduced due to secondary GC effects like improved cache locality.

Fig. 16 shows the overall execution times for each benchmark. These experiments are run on the unmodified VM. From a comparison between Figs. 14 and 16, it is clear to see that GC time is a small proportion of overall time. Fig. 17 shows the relative difference in overall times between the unmodified and modified VMs. A negative score indicates a speedup in the modified VM, whereas a positive score indicates a slow-down. There is a clear variation in performance, with four significant improvements at the minimum heap size.

From this small study, we can see that it is sometimes advantageous to employ the eager GC-after-optcomp policy, although sometimes it does not improve performance. Perhaps this strategy should be an adaptive VM option rather

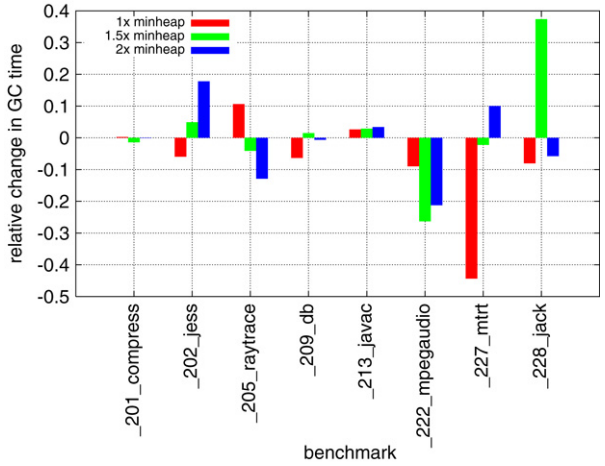

Fig. 15. Relative difference in GC times after VM modification, to force GC after optcomp.

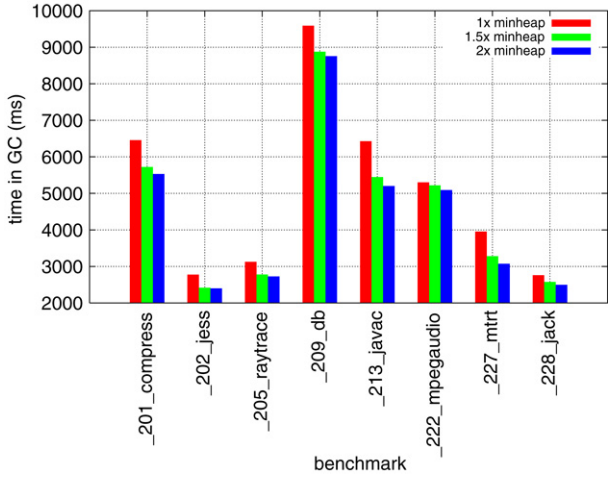

Fig. 16. Overall execution times for different benchmarks before VM modification.

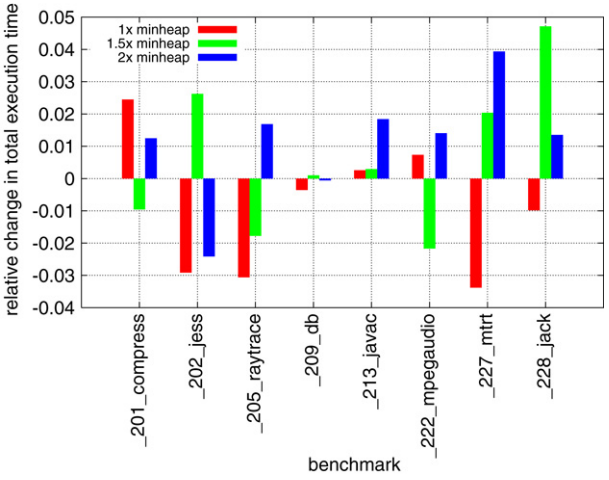

Fig. 17. Relative difference in overall execution times after VM modification, to force GC after optcomp.

than a hardwired choice, since it seems to depend on particular program characteristics. It should also depend on heap size configuration, growth policy and GC algorithm.

6. Related work

This paper is an extended version of a previous research [21]. The current paper improves on our earlier work in two ways:

(1) It provides a fuller treatment of the relationship between concepts and aspects (Sections 1.1 and 2.4).
(2) It uses concept-based profiling to investigate scheduling policies for garbage collection and optimizing compilation in Jikes RVM (Section 5).

6.1. Visualization systems

Hauswirth et al. [22] introduced the discipline of *vertical profiling* which involves monitoring events at all levels of abstraction (from hardware counters through virtual machine state to user-defined application-specific debugging statistics). Their system is built around Jikes RVM. It is able to correlate events at different abstraction levels in dynamic execution traces. They present some interesting case studies to explain performance anomalies in standard benchmarks. Our work focuses on user-defined high-level concepts, and how source code and dynamic execution traces are partitioned by concepts. Their work relies more on event-based counters at all levels of abstraction in dynamic execution traces.

GCspy [23] is an elegant visualization tool also incorporated with Jikes RVM. It is an extremely flexible tool for visualizing heaps and garbage collection behaviour. Our work examines processor utilization by source code concepts, rather than heap utilization by source code mutators.

Sefika et al. [24] introduced *architecture-oriented visualization*. They recognise that classes and methods are the base units of instrumentation and visualization, but they state that higher-level aggregates (which we term concepts) are more likely to be useful. They instrument methods in the memory management system of an experimental operating system. The methods are grouped into architectural units (concepts) and instrumentation is enabled or disabled for each concept. This allows efficient partial instrumentation on a per-concept basis, with a corresponding reduction in the dynamic trace data size. Our instrumentation is better in that it can operate at a finer granularity than method level. However our instrumentation cannot be selectively disabled, other than by re-assigning concepts to reduce the number of concept boundaries.

Sevitsky et al. [25] described a tool for analysing performance of Java programs using *execution slices*. An execution slice is a set of program elements that a user specifies to belong to the same category—again, this is a disguised concept. Their tool builds on the work of Jinsight [26] which creates a database for a Java program execution trace. Whereas Jinsight only operates on typical object-oriented structures like classes and methods, the tool by Sevitsky et al. handles compound execution slices composed of multiple classes and methods. They allow these execution slices to be selected manually or automatically. The automatic selection process is based on ranges of attribute values—for instance, method invocations may be characterized as slow, medium or fast based on their execution times.

Eng [27] presents a system for representing static and dynamic analysis information in an XML document framework. All Java source code entities are represented, and may be tagged with analysis results. This could be used for static representation of concept information, but it is not clear how the information could be extracted at runtime for the dynamic execution trace.

There are some Java visualization systems (for example, [28,29]) that instrument user code at each method entry and exit point to provide extremely detailed views of dynamic application behaviour. However these systems generate too much information to be useful for high-level comprehension purposes. In addition, they do not capture JVM activity.

Other Java visualization research projects (for example, [30,31]) instrument JVMs to dump out low-level dynamic execution information. However they have no facility for dealing with higher-level concept information. In principle it would be possible to reconstruct concept information from the lower-level traces in a post-processing stage, but this would cause unnecessarily complication, inefficiency and potential inaccuracy.

6.2. Eager garbage collection strategies

Buytaert et al. [20] gave a good overview of forced GC at potentially optimal points. They have profiling runs to determine optimal GC points based on heap usage statistics. They use the results of profiling to generate *hints* for the GC subsystem regarding when to initiate a collection. Wilson and Moher [32] appended GC onto long computational program phases, to minimise GC pause time in interactive programs. This is similar to our eager GC-after-optcomp approach. Both optcomp and GC reduce interactivity, so it is beneficial to combine these pauses whenever possible. Ding et al. [33] also exploited phase behaviour. They force GC at the beginning of certain program phases, gaining 40% execution time improvement. Their high-level phases are similar to our notion of concepts. They assume that most heap-allocated data is dead at phase transitions, and this assumption seems to be true for the single benchmark program they investigated. The behaviour of Jikes RVM is more variable and merits further investigation.

7. Concluding remarks

This paper has explored the dynamic analysis of concept information. This is a promising research area that has received little previous attention. We have outlined different techniques for embedding concept information in Java source code and dynamic execution traces. We have presented case studies of concept visualization and profiling. This high-level presentation of concept information seems to be appealingly intuitive. We have demonstrated the utility of this approach by harnessing the interaction between runtime compilation and garbage collection in the Jikes RVM adaptive runtime environment.

Until now, concepts have been a compile-time feature. They have been used for static analysis and program comprehension. The current work drives concept information through the compilation process from source code to dynamic execution trace, and makes use of the concept information in dynamic analyses. This follows the recent trend of retaining compile-time information until execution time. Consider typed assembly language, for instance [34].

During the course of this research project, we conceived a novel process which we term *feedback-directed concept assignment*. This involves: (1) selecting concepts; (2) assigning concepts to source code; (3) running the program; (4) checking results from dynamic analysis of concepts; and (5) using this information to repeat step (1). This is similar to feedback-directed (or profile-guided) compilation. In fact, this is how we reached the decision to examine both baseline and optimizing compilers separately in Section 4.1 rather than having a single compilation concept. We noticed that the single compilation concept (incorporating the activities of both baseline and optimizing compilers) was large, and did not correlate as well with the garbage collection concept. Once we split this concept into two, we observed that garbage collection follows optimizing compilation rather than baseline.

The process of feedback-directed concept assignment could be partially automated, given sufficient tool support. We envisage a system that allows users to specify the granularity of concepts in terms of source code (average number of lines per concept) or execution profile (average execution time percentage per concept) or both. The tool would process an initial concept assignment, execute the concept-annotated program and determine whether the user-specified requirements are met. If so, the tool indicates success. If not, the tool suggests a possible splitting of concepts, which the user has to approve or modify, then the tool re-assesses the concept assignment.

With regard to future work, we should incorporate the analyses and visualizations presented in this paper into an integrated development environment such as Eclipse. Further experience reports would be helpful, as we conduct more investigations with these tools. The addition of timestamps information to the phases visualization (Section 3.3) would make the comparison of different runs easier. We need to formulate other dynamic analyses in addition to concept proportions and phases. One possibility is *concept hotness*, which would record how the execution profile changes over time, with more or less time being spent executing different concepts. This kind of information is readily available for method-level analysis in Jikes RVM, but no one has extended it to higher-level abstractions.

Acknowledgements

The first author is employed as a research associate in the Jamaica project, which is funded by the EPSRC Portfolio Award GR/S61270/01. The small Java program described in Section 3 was written by Gavin Brown. We thank him for allowing us to analyse his program and report on the results. Finally we thank the *SCP* and *PPPJ '06* referees, together with many *PPPJ '06* conference attendees, for their thoughtful and helpful feedback on various iterations of this paper.

References

[1] T. Biggerstaff, B. Mitbander, D. Webster, Program understanding and the concept assignment problem, Communications of the ACM 37 (5) (1994) 72–82.

[2] N. Gold, K. Bennett, Hypothesis-based concept assignment in software maintenance, IEE Software 149 (4) (2002) 103–110.

[3] N. Gold, Hypothesis-based concept assignment to support software maintenance, in: Proceedings of the IEEE International Conference on Software Maintenance, 2001, pp. 545–548.

[4] N. Gold, M. Harman, D. Binkley, R. Hierons, Unifying program slicing and concept assignment for higher-level executable source code extraction, Software Practice and Experience 35 (10) (2005) 977–1006.

[5] J. Weidendorfer, Kcachegrind profiling visualization, 2005. See kcachegrind.sourceforge.net for details.

[6] N. Nethercote, J. Seward, Valgrind: A program supervision framework, Electronic Notes in Theoretical Computer Science 89 (2) (2003) 1–23.

[7] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, et al., The Jikes research virtual machine project: Building an open-source research community, IBM Systems Journal 44 (2) (2005) 399–417.

[8] A. Azevedo, A. Nicolau, J. Hummel, Java annotation-aware just-in-time (AJIT) compilation system, in: Proceedings of the ACM Conference on Java Grande, 1999, pp. 142–151.

[9] R.H.F. Brown, R.N. Horspool, Object-specific redundancy elimination techniques, in: Proceedings of Workshop on Implementation, Compilation, Optimization of Object-oriented Languages, Programs and Systems, 2006.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming., in: Proceedings of the 11th European Conference on Object-oriented Programming, 1997, pp. 220–242. URL: http://link.springer.de/link/service/series/0558/bibs/1241/12410220.htm.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of AspectJ, in: Proceedings of the 15th European Conference on Object-oriented Programming, 2001, pp. 327–353. URL: http://link.springer.de/link/service/series/0558/bibs/2072/20720327.htm.

[12] A. Popovici, T. Gross, G. Alonso, Dynamic weaving for aspect-oriented programming, in: Proceedings of the 1st International Conference on Aspect-oriented Software Development, 2002, pp. 141–147.

[13] M.H. Lipasti, C.B. Wilkerson, J.P. Shen, Value locality and load value prediction, in: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996, pp. 138–147.

[14] Y. Sazeides, J.E. Smith, The predictability of data values, in: Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture, 1997, pp. 248–258.

[15] J. Singer, G. Brown, Return value prediction meets information theory, Electronic Notes in Theoretical Computer Science 164 (3) (2006) 137–151.

[16] E. Duesterwald, C. Cascaval, S. Dwarkadas, Characterizing and predicting program behavior and its variability, in: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, 2003, pp. 220–231.

[17] A. Georges, D. Buytaert, L. Eeckhout, K. De Bosschere, Method-level phase behavior in Java workloads, in: Proceedings of the 19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 2004, pp. 270–287.

[18] P. Nagpurkar, C. Krintz, Visualization and analysis of phased behavior in Java programs, in: Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java, 2004, pp. 27–33.

[19] J. Singer, Concept assignment as a debugging technique for code generators, in: Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation, 2005, pp. 75–84.

[20] D. Buytaert, K. Venstermans, L. Eeckhout, K.D. Bosschere, GCH: Hints for triggering garbage collection, Transactions on High-performance Embedded Architectures and Compilers 1 (1) (2006) 52–72.

[21] J. Singer, C. Kirkham, Dynamic analysis of program concepts in Java, in: Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java, 2006, pp. 31–39.

[22] M. Hauswirth, P.F. Sweeney, A. Diwan, M. Hind, Vertical profiling: Understanding the behavior of object-oriented applications, in: Proceedings of the 19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 2004, pp. 251–269.

[23] T. Printezis, R. Jones, GCspy: An adaptable heap visualisation framework, in: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 2002, pp. 343–358.

[24] M. Sefika, A. Sane, R.H. Campbell, Architecture-oriented visualization, in: Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 1996, pp. 389–405.

[25] G. Sevitsky, W.D. Pauw, R. Konuru, An information exploration tool for performance analysis of Java programs, in: Proceedings of TOOLS Europe Conference, 2001.

[26] W.D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, Visualizing the execution of Java programs, in: Software Visualization, in: Lecture Notes in Computer Science, vol. 2269, 2002, pp. 151–162.

[27] D. Eng, Combining static and dynamic data in code visualization, in: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2002, pp. 43–50.

[28] S.P. Reiss, Visualizing Java in action, in: Proceedings of the IEEE Conference on Software Visualization, pp. 123–132.

[29] S.P. Reiss, M. Renieris, JOVE: Java as it happens, in: Proceedings of the ACM Symposium on Software Visualization, 2005, pp. 115–124.

[30] P. Dourish, J. Byttner, A visual virtual machine for Java programs: Exploration and early experiences, in: Proceedings of the ICDMS Workshop on Visual Computing, 2002.

[31] M. Golm, C. Wawersich, J. Baumann, M. Felser, J. Kleinöder, Understanding the performance of the Java operating system JX using visualization techniques, in: Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande, 2002, p. 230.

[32] P.R. Wilson, T.G. Moher, Design of the opportunistic garbage collector, in: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, 1989, pp. 23–35.
[33] C. Ding, C. Zhang, X. Shen, M. Ogihara, Gated memory control for memory monitoring, leak detection and garbage collection, in: Proceedings of the Workshop on Memory System Performance, 2005, pp. 62–67.
[34] G. Morrisett, D. Walker, K. Crary, N. Glew, From system F to typed assembly language, ACM Transactions on Programming Languages and Systems 21 (3) (1999) 527–568.